Real-Time Ticks for Synchronous Programming

Reinhard von Hanxleden, Timothy Bourke, Alain Girault

Real-Time Ticks for Synchronous Programming

Reinhard von Hanxleden*, Timothy Bourke†, Alain Girault‡
*Department of Computer Science, Kiel University, Kiel, Germany, E-mail: rvh@informatik.uni-kiel.de
†INRIA and École Normale Supérieure, PSL Research University, Paris, France, E-mail: Timothy.Bourke@inria.fr
‡Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France , E-mail: alain.girault@inria.fr

***Abstract*—We address the problem of synchronous programs that cannot be easily executed in a classical time-triggered or event-triggered execution loop. We propose a novel approach, referred to as *dynamic ticks*, that reconciles the semantic timing abstraction of the synchronous approach with the desire to give the application fine-grained control over its real-time behavior. The main idea is to allow the application to dynamically specify its own wake-up times rather than ceding their control to the environment. As we illustrate in this paper, synchronous languages such as Esterel are already well equipped for this; no language extensions are needed. All that is required is a rather minor adjustment of the way the tick function is called.**

***Index Terms*—Real-time systems, reactive systems, synchronous languages, physical time, Esterel**

I. Introduction

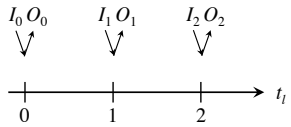
Reactive systems are characterized by regularly interacting with their environment in a cyclic manner. In each cycle—also known as a *tick* or *instant*—the system 1) collects *inputs* provided by sensors, 2) computes a *reaction*, in a *tick function* (also called a *step function*), and 3) provides *outputs* to actuators. This generic pattern is illustrated in Lst. 1.

```
int main() {
  int notDone;
  reset();
  do {
    get_inputs();
    notDone = tick();
    set_outputs();
  } while (notDone);
  return 0;
}
```

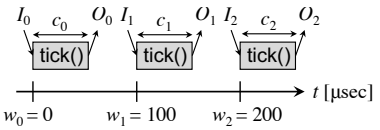
Listing 1. Generic reactive C code

Synchronous languages conceptually assume that outputs are synchronous with inputs [3]. They thus abstract from the time it takes to compute a reaction. This abstraction facilitates the definition of a deterministic semantics, even when the computation of a reaction involves concurrency and shared data. This determinism is a major strength of synchronous languages. Other languages based on asynchronous threads, like Java [12], for instance, are not deterministic: the result of a reaction may depend on scheduling choices made by the execution environment.

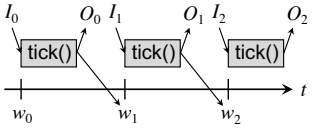
Esterel does not have an inbuilt notion of physical time, but instead employs the more general notion of *multiform time*. The repetition of any event, such as "a car has driven 1 m" or "one second has passed" defines a notion of time and order [5]. The synchronous abstraction engenders a notion of *discrete logical time*, as illustrated in Fig. 1a. Logical time is shared by concurrent threads. There is thus a clear semantic concept of temporal order across threads, which defines whether, for example, two accesses to a shared variable occur in the same tick or in different ticks. The order does not depend on the time it takes on a particular execution platform to compute a reaction for a particular system state and set of inputs.



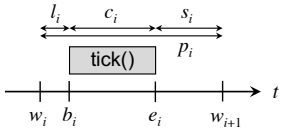
(a) Logical time: time is discretized into logical ticks 0, 1, ... Input I_i is synchronous with output O_i, the reaction time is abstracted to be 0.



(b) Physical time: the computation of the ith reaction, corresponding to logical tick i and the ith call of the tick function, begins at *wake-up time* w_i. Inputs are read at the beginning of the computation, outputs are written at the end of the computation. The wake-up times are controlled by the environment.



(c) Dynamic physical time: the wake-up times are controlled by the tick function.



(d) Detailed physical time: for tick i, lag l_i expresses how long after w_i the tick function is called, typically due to timer imprecisions; compute time c_i is the time it takes to compute the tick function; slack s_i measures the time remaining after the tick function completes until the next tick function should be called. The *period* p_i, the duration of the ith tick, is defined as $w_{i+1} - w_i$. Ticks begin at $b_i = w_i + l_i$, and end at $e_i = b_i + c_i$.

Fig. 1. Different timing abstractions.

In reality, the computation of a reaction does take time. This is reflected in the *physical time view* shown in Fig. 1b. Each tick i takes some time c_i to compute outputs O_i, based on the internal state and inputs I_i. Furthermore, logical time is mapped to physical time by associating tick i with the *wake-up time* w_i. Here we measure w_i relative to the start time of the reactive system. Thus the wake-up time of the initial tick is $w_o = 0$.

A. Execution modes of synchronous programs

Traditionally, three execution modes are distinguished:

1) *Free-running*: As soon as the computation of a reaction finishes, the next reaction is started. The scheme shown in Lst. 1 falls into this category.

2) *Event-triggered:* A reaction is triggered by events from the environment, for example, the pressing of a button.
3) *Time-triggered:* Reactions are started at regular intervals, for example, every $100\,\mu\text{sec}$, as shown in Fig. 1b.

For simple periodic systems, the semantic abstraction of physical time into logical time does not pose any difficulties. However, embedded real-time systems often have fairly intricate timing characteristics that do not easily map to any of the aforementioned execution modes. Consider for instance a two-wheel robot, where each wheel is individually powered by a stepper motor. The speed of the motors depends on the periods of the motor actuators. To make a turn, the motors should be allowed to run at different speeds with the "outside" wheel turning faster. The robot controller thus needs fine-grained control of when actuator outputs occur.

Note that in all the execution modes described above, the wake-up times are controlled by the environment, not by the application. This is, however, not intrinsically necessary. This is precisely the main idea of this paper, to give the program control over its wake-up times, as illustrated in Fig. 1c.

B. Expressing time in synchronous languages

Several techniques are used to relate the logical time of synchronous programs to the physical time of their environment. We briefly recall them here based on an earlier survey [8].

The Esterel pause statement blocks the execution of a thread until the next reaction. Its relationship to physical time is simple in the time-triggered mode, it gives a physical delay of one period, but it is less clear-cut in the free-running and event-triggered modes. In the former it depends on execution time, and in the latter, on external events.

More typically, delays are expressed by introducing timing inputs and counting their occurrences. For instance, to wait for 4 sec, one could introduce a signal called SEC and then delay with await 4 SEC. This approach is compositional in that, unlike pause, it is unaffected by the addition of other threads or inputs to the program, and it combines reasonably well with suspension. The disadvantages are that a direct implementation "busy waits", that is, a program may be executed frequently just to decrement counters, and its Worst Case Execution Time (WCET) must be less than the period of the fastest timing input. Furthermore, the fastest timing input may not be needed in all program modes. Using more granular timing inputs mitigates these problems but may reduce precision since the inputs are not synchronized with await statements. For instance, in the program

```
await I; await 4 SEC; emit O,
```

the physical delay between an occurrence of I and the emission of O is in the interval $(3, 4]$. It depends on when the next SEC occurs relative to the occurrence of the input I.

Another technique is to link external one-shot timers to certain input and output signals. For instance, to wait 100 ms, one could write:

```
emit START_MSTIMER1(100); await MSTIMER1,
```

where the first command emits a valued signal to a particular millisecond timer and the second awaits the signal that marks timer expiry. In contrast to timing inputs, the delay is relative to the triggering statement and finer granularities are less problematic, essentially since the delay is performed in dedicated hardware. One shot timers are a standard technique in embedded software, and they are especially well suited to the event-driven mode. That said, the approach presented above is not as abstract as might be hoped and the use of timers in this way introduces implementation details and specific platform requirements into the specification. Properly combining timers with the abort and suspend constructs requires the support of the language runtime [4].

A simple idea for separating the specification of a delay from the implementation mechanism is to introduce a macro statement [8] that is later rewritten for a given platform using any of the previously mentioned techniques. However, this idea has not been rigorously developed.

C. Contributions and Outline

We propose a novel approach, referred to as *dynamic ticks*, that reconciles the semantic timing abstraction of the synchronous approach with the desire to give the application fine-grained control over its real-time behavior. The main idea is to allow the application to dynamically specify its own wake-up times rather than ceding their control to the environment. As we illustrate in this paper, synchronous languages such as Esterel are already well equipped for this; no language extensions are needed. All that is required is a rather minor adjustment of the way the tick function is called.

We illustrate our proposal with a running example, RACE, introduced in the next section. Sec. III then presents the dynamic tick proposal in detail. Sec. IV discusses further concepts that build on dynamic ticks, followed by a brief discussion of related work in Sec. V before concluding.

II. The RACE Example

The RACE example is specified as follows: two threads, writeX ("Thread 1") and readX ("Thread 2"), first run "asynchronously," by performing time-outs of differing durations and fine granularity, but then synchronize again to simultaneously access a shared variable wX. Specifically, writeX performs three time-outs of duration wait1_usec, where wait1_usec is some random duration below one second, and readX performs two time-outs of duration wait2_usec, chosen to be 1.5 times that of wait1_usec.

The functionality of RACE is admittedly rather artificial, but represents a scenario that is non-trivial for both standard synchronous programming and for C-/Posix-/Java-style threads alike, for similar reasons as the robot example sketched earlier. From the synchronous perspective, RACE does not fit any of the operation modes listed in Sec. I-A, as they do not give the application direct control over the wake-up times. For standard threads, they pose the difficulty that, as the name of the example suggests, there is a race condition between the accesses of writeX and readX to wX.

```
1  %%%%%%%% RACE %%%%%%%%
2  % Main module
3  % WRITE_X and READ_X always terminate at the _same_ tick,
4  % without inducing a data race.
5
6  module RACE:
7
8  function min(integer, integer) : integer;
9  function rand() : integer;
10 function print_int (string , integer) : integer;
11
12 input current_wall_usec : integer;
13 input prev_tick_end_usec : integer;
14 output wake_usec : combine integer with min;
15
16 var d : integer in          % Dummy variable, for calling print_int ()
17 signal W_X,                 % Shared variable
18   R_X : integer,            % Indicates read of W_X
19   current_usec : integer,   % Simulated time
20   wait1_usec : integer,     % Timeouts for Thread 1
21   wait2_usec : integer      % Timeouts for Thread 2
22 in
23 [
24   run TIME;
25 ||
26   loop
27     % Wait for 0...999998 usec or 0...1499997 usec
28     emit wait1_usec((rand() mod 500000) * 2);
29     emit wait2_usec(?wait1_usec / 2 * 3);
30
31     [ run WRITE_X || run READ_X ];
32     d := print_int ("====< R_X = %d >====\n", ?R_X);
33   end
34 ]
35 end signal
36 end var
37 end module
```

Listing 2. The Esterel module RACE, which computes the timeouts to be performed by WRITE_X and READ_X and then concurrently instantiates these threads. When the threads terminate, it displays the value of R_X. Concurrently, RACE instantiates TIME.

```
1  %%%%%%%% WRITE_X, Thread 1 %%%%%%%%
2  % Three timeouts, then emit W_X
3
4  module WRITE_X:
5
6  function min(integer, integer) : integer;
7
8  input current_usec : integer;
9  input wait1_usec : integer;
10 output wake_usec : combine integer with min;
11 output W_X;
12
13 run PAUSE_USEC [ constant 1 / id; signal wait1_usec / wait_usec ];
14 run PAUSE_USEC [ constant 1 / id; signal wait1_usec / wait_usec ];
15 run PAUSE_USEC [ constant 1 / id; signal wait1_usec / wait_usec ];
16 emit W_X;
17 end module
18
19
20 %%%%%%%% READ_X, Thread 2 %%%%%%%%
21 % Two timeouts, then read W_X
22
23 module READ_X:
24
25 function min(integer, integer) : integer;
26
27 input current_usec : integer;
28 input wait2_usec : integer;
29 input W_X;
30 output wake_usec : combine integer with min;
31 output R_X : integer;
32
33 run PAUSE_USEC [ constant 2 / id; signal wait2_usec / wait_usec ];
34 run PAUSE_USEC [ constant 2 / id; signal wait2_usec / wait_usec ];
35 present W_X then emit R_X(1) else emit R_X(0) end;
36 end module
```

Listing 3. The Esterel modules WRITE_X ("Thread 1") and READ_X ("Thread 2"), which do some time-outs and then both access the shared signal W_X. Depending on whether W_X is present or not when it is tested by READ_X, the valued signal R_X is set to 1 or 0.

In the following, we first discuss an Esterel version of RACE, followed by a C version.

A. The Esterel Variant of RACE

The Esterel module RACE, shown in Lst. 2, concurrently instantiates the modules WRITE_X and READ_X, shown in Lst. 3. These modules perform time-outs using the PAUSE_USEC module, shown in Lst. 4 and discussed further in Sec. III. Lst. 5 shows the Esterel TIME module, which manages the simulated time current_usec by initializing it to 0 (line 13) in the initial tick and, in subsequent ticks, setting it to the wake-up time computed in the previous tick (line 28). It also keeps track of the logical time, by incrementing tickCnt (line 29). Lst. 7 shows time-related utility functions that are used by both the Esterel and the C versions of RACE.

The Esterel semantics, together with our implementation of time-outs, guarantees that Threads 1 and 2 terminate in exactly the same tick, and that communication via the shared signal W_X obeys Esterel's deterministic emit-before-test semantics. Thus the printed value of R_X (see Lst. 2, line 32) is always 1.

In C, the RACE example could also be made deterministic with an emit-before-test behavior, but at the significant cost of some form of barrier synchronization. Such an approach

```
1  ----< Tick 0 >----
2  Simulated time: 0 usec
3  Lag: 0 usec
4  Last compute time: 0 usec
5  Last slack: 0 usec
6  Thread 1 waits 33614 usec...
7  Thread 2 waits 50421 usec...
8  ----< Tick 1 >----
9  Simulated time: 33614 usec
10 Lag: 3369 usec
11 Last compute time: 157 usec
12 Last slack: 33457 usec
13 Thread 1 waits 33614 usec...
14 ----< Tick 2 >----
15 Simulated time: 50421 usec
16 Lag: 779 usec
17 Last compute time: 51 usec
18 Last slack: 13387 usec
19 Thread 2 waits 50421 usec...
20 ----< Tick 3 >----
21 Simulated time: 67228 usec
22 Lag: 4058 usec
23 Last compute time: 284 usec
24 Last slack: 15744 usec
25 Thread 1 waits 33614 usec...
26 ----< Tick 4 >----
27 Simulated time: 100842 usec
28 Lag: 5054 usec
29 Last compute time: 53 usec
30 Last slack: 29503 usec
31 ====< R_X = 1 >====
32 Thread 1 waits 950498 usec...
33 Thread 2 waits 1425747 usec...
34 ----< Tick 5 >----
35 Simulated time: 1051340 usec
36 Lag: 1800 usec
37 Last compute time: 106 usec
38 Last slack: 945338 usec
39 Thread 1 waits 950498 usec...
40 ----< Tick 6 >----
41 Simulated time: 1526589 usec
42 Lag: 2666 usec
43 Last compute time: 69 usec
44 Last slack: 473380 usec
45 Thread 2 waits 1425747 usec...
46 ----< Tick 7 >----
47 Simulated time: 2001838 usec
48 Lag: 2641 usec
49 Last compute time: 58 usec
50 Last slack: 472525 usec
51 Thread 1 waits 950498 usec...
52 ----< Tick 8 >----
53 Simulated time: 2952336 usec
54 Lag: 2033 usec
55 Last compute time: 57 usec
56 Last slack: 947800 usec
57 ====< R_X = 1 >====
```

Fig. 2. Sample trace of the Esterel version of RACE.

```
1  %%%%%%%% PAUSE_USEC %%%%%%%%
2  % Pause for wait_usec microseconds
3
4  module PAUSE_USEC:
5
6  function min(integer, integer) : integer;
7  function print_2int(string, integer, integer) : integer;
8
9  input current_usec : integer;    % Simulated time
10 input wait_usec : integer;       % Time of delay
11 output wake_usec : combine integer with min; % Time of next wake up
12 constant id : integer;           % id of calling thread (1 or 2)
13 var my_wake_usec : integer,      % Local, persistent copy of wake_usec
14   d : integer
15 in
16 d := print_2int("Thread %d waits %d usec...\n", id, ?wait_usec);
17
18 % Compute physical time when PAUSE_USEC should terminate
19 my_wake_usec := ?current_usec + ?wait_usec;
20
21 % Loop until current_usec = my_wake_usec
22 trap done in
23   loop
24     emit wake_usec(my_wake_usec);
25     pause;
26     if ?current_usec = my_wake_usec then
27       exit done;
28     end if;
29   end loop
30 end trap
31 end var
32 end module
```

Listing 4. The Esterel PAUSE_USEC module, which pauses for a time-out of wait_usec µsec. It first computes, in my_wake_usec (line 19), when the time-out will have expired. Then it announces this via wake_usec (line 24), which is globally combined with other pending time-outs through the min function (line 11). This is repeated until the actual time (current_usec) has reached my_wake_usec (lines 26/27).

```
1  %%%%%%%% TIME %%%%%%%%
2  % Manage simulated time, do some logging
3
4  module TIME:
5
6  function min(integer, integer) : integer;
7  function rand() : integer;
8  function print_int(string, integer) : integer;
9
10 input wake_usec : integer;            % Time of next wake up
11 input prev_tick_end_usec : integer;   % Previous tick completion time
12 input current_wall_usec : integer;    % Real time
13 output current_usec := 0 : integer;   % Simulated time
14
15 var d : integer in  % dummy variable to facilitate print calls
16 signal tickCnt := 0 : integer in
17 loop
18   d := print_int("----< Tick %d >----\n", ?tickCnt);
19   d := print_int("Simulated time: %d usec\n",
20        ?current_usec);
21   d := print_int("Lag: %d usec\n",
22        ?current_wall_usec - ?current_usec);
23   d := print_int("Last compute time: %d usec\n",
24        ?prev_tick_end_usec - pre(?current_wall_usec));
25   d := print_int("Last slack: %d usec\n",
26        ?current_usec - ?prev_tick_end_usec);
27   pause;
28   emit current_usec(pre(?wake_usec)); % Advance simulated time
29   emit tickCnt(pre(?tickCnt) + 1);    % Advance logical time
30 end
31 end signal
32 end var
33 end module
```

Listing 5. The Esterel TIME module, which manages simulated time and logical time.

```
1  ----------                          27 ----------
2  Thread 1 sim. time: 0 usec          28 Thread 1 sim. time: 100842 usec
3  ----------                          29 Thread 1 lag: 2719 usec
4  Thread 2 sim. time: 0 usec          30 Thread 1 slack: 947779 usec
5  Thread 1 lag: 117 usec              31 Thread 1 waits 950498 usec...
6  Thread 2 lag: 149 usec              32 ----------
7  Thread 1 slack: 33497 usec          33 Thread 2 sim. time: 100842 usec
8  Thread 2 slack: 50272 usec          34 Thread 2 lag: 2760 usec
9  Thread 1 waits 33614 usec...        35 Thread 2 slack: 1422987 usec
10 Thread 2 waits 50421 usec...        36 Thread 2 waits 1425747 usec...
11 ----------                          37 ----------
12 Thread 1 sim. time: 33614 usec      38 Thread 1 sim. time: 1051340 usec
13 Thread 1 lag: 2028 usec             39 Thread 1 lag: 5207 usec
14 Thread 1 slack: 31586 usec          40 Thread 1 slack: 945291 usec
15 Thread 1 waits 33614 usec...        41 Thread 1 waits 950498 usec...
16 ----------                          42 ----------
17 Thread 2 sim. time: 50421 usec      43 Thread 2 sim. time: 1526589 usec
18 Thread 2 lag: 1873 usec             44 Thread 2 lag: 5194 usec
19 Thread 2 slack: 48548 usec          45 Thread 2 slack: 1420553 usec
20 Thread 2 waits 50421 usec...        46 Thread 2 waits 1425747 usec...
21 ----------                          47 ----------
22 Thread 1 sim. time: 67228 usec      48 Thread 1 sim. time: 2001838 usec
23 Thread 1 lag: 1800 usec             49 Thread 1 lag: 538 usec
24 Thread 1 slack: 31814 usec          50 Thread 1 slack: 949960 usec
25 Thread 1 waits 33614 usec...        51 Thread 1 waits 950498 usec...
26 =========< rX = 0 >=========       52 =========< rX = 1 >=========
```

Fig. 3. Sample trace of the C version of RACE.

has been implemented in PRET-C for single cores [1] and in ForeC for multicores [18].

An example output trace is shown in Figure 2. This shows the first two iterations of the outer loop. There are significant variations in the lag, between 779 µsec (Tick 2) and 5 054 µsec (Tick 4). This is not too surprising, since the platform on which the tests were performed (1.6 GHz Intel Core i5 running MacOS 10.12.4) is not an embedded real-time system. However, despite these variations of *when* the system does actually react, it is deterministic in *what* it produces. The trace also shows the compute times for the tick functions, which in this rather trivial example are well below the average lag times.

B. The C Variant of RACE

The main() program shown in Lst. 8 instantiates Threads 1 and 2 using the writeX() and readX() functions. The timeouts are performed using the function pause_usec(). This, like the Esterel version of RACE, relies on the function microsleep(), shown in Lst. 7.

An example output trace is shown in Figure 3. The sequence of randomly generated time-outs is the same as for the Esterel version, since we use the same system calls to the pseudo-random number generator and do not initialize it. However, as seen in lines 26 and 52 of the trace, the write-read race on wX is resolved differently. In the first iteration of the while-loop, the wX was written by writeX() *after* it was read by readX(), resulting in rX = 0, while in the second iteration, wX was written by writeX() *before* it was read by readX(), resulting in rX = 1. This behavior can differ from run to run and is not predictable. This non-determinism is typical for POSIX

```
int main()
{
  int notDone, prev_tick_end_usec = 0;

  //srand(time(NULL)); // Init random seed
  RACE_reset();      // Reset automaton
  time_reset();      // Initialize time

  // Loop until tick function terminates
  do {
    // Set inputs
    RACE_I_current_wall_usec(get_current_wall_usec());
    RACE_I_prev_tick_end_usec(prev_tick_end_usec);

    notDone = RACE(); // Call tick function
    prev_tick_end_usec = get_current_wall_usec();

    // Wait until wake_usec
    microsleep(wake_usec - prev_tick_end_usec);
  } while (notDone);

  return 0;
}
```

Listing 6. Host language context for the Esterel RACE example. It provides the main function that repeatedly calls the tick function RACE(). Not shown here are the definitions of the min() combination function used for computing the wake-up time wake_usec, the function RACE_O_wake_usec() that communicates the wake_usec output of the tick function, and the printing utility functions print_int() and print_2int().

```
struct timespec time_wall_init;   // Start time

/**** Initialize time ****/
void time_reset() {
  clock_gettime(CLOCK_REALTIME, &time_wall_init);
}

/**** Get usecs since start ****/
int get_current_wall_usec() {
  struct timespec time_wall;

  clock_gettime(CLOCK_REALTIME, &time_wall);
  int current_wall_sec = time_wall.tv_sec - time_wall_init.tv_sec;
  long current_wall_nsec = time_wall.tv_nsec - time_wall_init.tv_nsec;
  int current_wall_usec = current_wall_sec * 1e6 + current_wall_nsec /
      1000;

  return current_wall_usec;
}

/**** Sleep for sleep_usec ****/
void microsleep(int sleep_usec) {
  struct timespec time_req, time_rest;
  if (sleep_usec > 0) {
    time_req.tv_sec  = sleep_usec / 1e6;
    time_req.tv_nsec = 1000 * (sleep_usec % 1000000);
    nanosleep(&time_req, &time_rest);
  }
}
```

Listing 7. C utility functions for dealing with time, used by both the C and the Esterel version of RACE. time_reset() initializes time_wall_init. get_current_wall_usec() returns the time passed since calling time_reset(). microsleep() sleeps for sleep_usec µsec.

threads used by C and also for threads in Java. The trace also shows different interleavings of the outputs produced by Threads 1 and 2 at the beginning of the while loop. In this run, at least in both iterations of the while loop, Thread 1 happened to be first to print "Thread i waits ...," but that is also non-deterministic; in some runs Thread 2 prints first.

```
// Simulated time for Thread 1/2; we don't use index 0
int current_usec[3] = { 0, 0, 0 };
int wX, rX; // Shared variable

/**** Pausing routine ****/
void pause_usec(int id, int wait_usec) {
  int current_wall_usec = get_current_wall_usec();

  printf ("----------------------------\
    nThread %d simulated time: %d usec\n", id, current_usec[id]);
  printf ("Thread %d lag: %d usec\n", id, current_wall_usec - current_usec
    [id]);
  current_usec[id] += wait_usec;
  printf ("Thread %d slack: %d usec\n", id, current_usec[id] -
    current_wall_usec);
  printf ("Thread %d waits %d usec...\n", id, wait_usec);
  microsleep(current_usec[id] - current_wall_usec);
}

/**** writeX, Thread 1 ****/
// Three timeouts, then emit wX
void *writeX(void* arg) {
  int wait_usec = *(int*) arg;
  wX = 0;
  pause_usec(1, wait_usec);
  pause_usec(1, wait_usec);
  pause_usec(1, wait_usec);
  wX = 1;
  return NULL;
}

/**** readX, Thread 2 ****/
// Two timeouts, then read wX
void *readX(void* arg) {
  int wait_usec = *(int*) arg;
  pause_usec(2, wait_usec);
  pause_usec(2, wait_usec);
  rX = wX;
  return NULL;
}

/**** Main Thread ****/
int main() {
  pthread_t thread1, thread2;
  int wait1_usec, wait2_usec;

  //srand(time(NULL)); // Init random seed
  time_reset();          // Init wall clock time

  // Loop until "rX = wX" occurs after "wX = 1"
  while (1) {
    // Timeout 1, up to 999998 usec
    wait1_usec = (rand() % 500000) * 2;

    // Timeout 2, up to 1499997 usec
    wait2_usec = wait1_usec / 2 * 3;

    // Create threads
    pthread_create(&thread1, NULL, &writeX, &wait1_usec);
    pthread_create(&thread2, NULL, &readX, &wait2_usec);

    // Wait for threads
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf ("=========< rX = %d >=========\n", rX);
  };

  return 1;
}
```

Listing 8. C version of RACE, without include's and external function declarations

There are again significant variations of the lag, between $117\,\mu\mathrm{sec}$ and $5\,207\,\mu\mathrm{sec}$, which we explain with imprecisions of the calls to nanosleep().

To make the behavior of RACE deterministic, we can either resort to explicit synchronizations using semaphores and the like, or we can turn to synchronous programming, as we do in the Esterel version.

III. Dynamic ticks in Esterel

The idea of *dynamic ticks* is to give an Esterel program access to physical time by slightly extending the standard tick function loop shown in Lst. 1. Most importantly, the tick function computes the wake-up time for the next tick. Technically, this is achieved by adding an output, called wake_usec in our example implementation, to the Esterel tick function, see Lst. 6. We also inform the tick function about 1) when the tick function has been called, i. e., the tick begin time b_i, and 2) when the previous tick function call has finished, i. e., the tick end time e_{i-1}; note that e_i is not known when the tick function is called. At the initial tick, we set $e_{-1} = 0$. This is implemented by two inputs, current_wall_usec and prev_tick_end_usec, see Lst. 6, lines 12–13.

The interesting question now is how to deal with concurrent time-outs, especially if they have different durations, as is the case in the RACE example. One option would be to ask the environment to provide as many timers as may be active simultaneously, but this violates the interface described above in which only one wake-up time is specified. We resolve this at the application level, by specifying the wake-up time as the time of the most immediate time-out. In Esterel, we can achieve this quite easily by using a shared valued signal wake_usec that is combined with the min function, see line 11 in Lst. 4. Each time-out then has to check whether it has expired, which is the case when the current time has reached the time of the time-out, see Lst. 4, line 26. If so, then the time-out expires and the PAUSE_USEC module terminates. If not, then the time-out is re-asserted (line 24).

A. Determinism

Esterel is deterministic in that for a given sequence of inputs, the program produces a deterministic sequence of outputs. This still holds for Esterel with dynamic ticks proposed here, however, now the inputs include b_i and e_{i-1}.

This means that the lag and compute times, which are platform dependent and manifest themselves in b_i and e_{i-1}, may influence the program behavior, as in C. However, there is still no non-determinism due to race conditions or run-time scheduling within a reaction, unlike in a C program that requires additional mechanisms such as barrier synchronization to rule out race conditions.

Non-determinism can also arise at the interface between the synchronous Esterel program and the asynchronous environment. In the case where the timeouts are based on the wall-clock time, the fact that a larger lag may cause two external events to be simultaneous while they would occur at different ticks if the lag were smaller. This non-determinism at the synchronous-asynchronous interface is unavoidable [2].

B. Wall clock time vs. simulated time

In our implementation, we use the "simulated time" current_usec as the current time and perform the time-out check by checking for equality between current_usec and the time of the time-out, see line 26 in Lst. 4.

Alternatively, time-outs could also check the real wall clock time current_wall_usec provided by the environment, but in that case we should not test for strict equality, and instead should use a "greater-or-equal-to" comparison.

A difference is that time-outs based on simulated time put a strict order on time-outs in that time-outs that are scheduled to expire at different times will expire in different ticks. As alluded to in Sec. III-A, time-outs based on wall clock time might expire in the same tick if the difference between their expiration times is larger than the current lag. Depending on the application, this may or may not be a problem. Conversely, time-outs based on simulated time may cause the lag to become arbitrarily long if time-outs are continually shorter than tick computation times.

In both variants we still rule out race conditions, and we guarantee that concurrent threads that are waiting for the same timer expiration time will consider their timers to be expired in the same tick.

C. Efficiency Considerations

Dynamic ticks are, primarily, a semantic concept for augmenting logical ticks with physical time. For illustration, we present a concrete realization based on Esterel, but dynamic ticks can be implemented in very different ways. One natural question arising is the efficiency of the realization. For example, the Esterel realization of a timer shown in the PAUSE_USEC module (Lst. 4) conceptually performs a "busy wait" until the wake-up time is reached. This may appear quite inefficient, as each thread must individually check whether its timer has expired or not, and the overall time this takes increases linearly with the number of pending time outs. Other realization schemes for dynamic ticks would be possible, for example to maintain an ordered queue for all pending time outs as is common in event-driven programming. However, for a "reasonable" number of pending time outs, the scheme sketched here for Esterel does not have to be all that expensive after all, since thread context switches are basically resolved at compile time and, e. g., do not incur any system calls.

IV. Building on Dynamic Ticks

So far, we considered the simple case of an application that controls when it wants to be woken up. We also illustrated how concurrent threads with different time-outs can be managed at the application level, in a simple, modular fashion. However, there are numerous questions and opportunities that arise from this and that we now briefly discuss.

A. Wake-Up Times and Outputs

As shown in Fig. 1c, the wake-up time w_i specifies when tick i should be started. However, as illustrated in Fig. 1d, the exact production time of output O_i also depends on the

lag l_i, compute time c_i, and the way that the outputs computed by a reaction are actually propagated to the environment. For example, the procedural interface of Esterel specifies that inputs are provided to the tick function by calling specific functions before calling the tick function, so the sampling of inputs happens strictly before the computation of the reaction. Outputs, however, are made available by the tick function as it executes, and not at the very end. Of course, one could employ a buffering scheme that makes outputs available to the environment only after the tick function has terminated, but this is not the default. These factors should be considered when choosing wake-up times.

B. Timing Analysis, Timing Overruns

A common question is whether a system is "fast enough." The exact meaning of this question depends on the application, but what we generally wish to avoid is a *timing overrun*, meaning that a tick computation finishes after the wake-up time specified for the next tick. In other words, the slack s should never become negative. As illustrated in Fig. 1d, it is $s_i = p_i - (l_i + c_i) = w_{i+1} - w_i - l_i - c_i$. First considering the lag l, to achieve $s_i \geq 0$, we clearly would like l to be as small as possible. What is achievable with respect to timer precisions depends on the deployment platform and is not considered further now. Assuming the lag to be minimal, i. e., $l_i = 0$, we want to ensure $p_i \geq c_i$. In other words, the period should not be shorter than the WCET of the tick function.

Thus, an interesting and non-trivial issue is to determine the periods that may arise in a system, and, in particular, the shortest one. In the trace of the RACE example shown in Fig. 2, the shortest period occurs between Tick 1 and Tick 2, with $p_1 = w_2 - w_1 = 50\,421\,\mu\text{sec} - 33\,614\,\mu\text{sec} = 16\,807\,\mu\text{sec}$. In this trace, that period is long enough to always have a positive slack. However, this is not always guaranteed, as the randomly generated wake-up times may be arbitrarily small. While the random behavior of RACE is, admittedly, rather artificial, there may also be other, more realistic examples where periods may become arbitrarily small. Consider again the two-wheel robot example. To initially drive straight, the motors run synchronously with each other. However, to take a slight turn, the wheel motor rotations should be slightly shifted from each other, resulting in an arbitrarily short period between the control outputs for the left and right motors.

The dynamic tick concept is very flexible, but has the disadvantage that the possible periods cannot be predicted in general. Safe approximations are, however, often feasible. For example, if the requested wait times of all threads are statically known, their gcd is a safe lower bound on the periods that may result from arbitrary interleavings of the wait times.

A related question is how to deal with timing overruns. This again depends on the application. E. g., in the robot example timing overruns may be acceptable if they only occur rarely. Other applications may, e. g., require exception handling mechanisms if a timing overrun is detected, or they may at least have to somehow record such timing overruns. Again, the flexible tick interface is agnostic here and gives the freedom to handle timing overruns at the application level.

C. Time-Triggered and Event-Triggered Systems

The timeout of a thread is robust in that there is no harm in inserting an arbitrary number of logical ticks, for whatever reason, before the timer should expire. Thus, dynamic ticks work seamlessly with the event-triggered, time-triggered, and free-running execution modes. Besides, an application can make itself time-triggered by simply including a "time-trigger thread" that pauses a fixed duration in an infinite loop. The fixed duration could be relative to either the last time-out of the time-trigger thread or to the last wake-up time.

D. Mixed-Criticality Systems

If we cannot guarantee that periods are never shorter than the WCET, it might be possible to divide the application into parts that are *critical* and must always be executed, and other parts that are *non-critical* and that are only executed if the current period is long enough. Similarly, we may divide a system into different modes, e. g., a *fast mode*, which produces low-quality outputs, and a *slow mode* with higher-quality results. An application may use these concepts, which are already well-established in embedded systems design, in conjunction with the concept of dynamic ticks to minimize the lag. This may be done in a purely measurement-based way, based on observed compute times, or in a more analytical fashion based on WCET-values for, e. g., the fast mode and the slow mode. A system may, e. g., start in the slow mode and switch to the fast mode if the slack becomes negative.

V. Related Work

There have been several proposals for introducing continuous-time features into synchronous programs. For instance, adding non-deterministic pauses statements to model platform constraints in Esterel [6], interpreting programs with timeouts as timed automata for verifying quantitative properties [11], or, more recently, introducing the possibility to model and simulate the continuous environment of a reactive system [7]. However, these approaches do not specifically treat the interface with the physical environment.

The concept of a wake-up time is similar to the deadline instruction used in the PRET approach [13]. There, the deadline instruction is used to schedule shared variable accesses by concurrent threads, for example, by having one thread periodically produce data ahead of another consumer thread that runs at the same rate but with a constant delay. As pointed out elsewhere [17], this use of the deadline instruction semantically corresponds to a pause instruction, the main difference being that, in synchronous programming, the scheduling is not done at the application level, but by the compiler. Our proposal of a wake-up time could, in principle, also be used to schedule shared data accesses, as in the PRET approach. However, the main motivation is somewhat different, namely to provide an application with direct control of its timing behavior with respect to the physical environment.

The PTIDES (Programming Temporally Integrated Distributed Embedded Systems) system addresses the design and implementation of distributed real-time embedded systems [9]. The nodes in a PTIDES system are synchronized to provide a global time base and signal values are paired with time stamps. Time stamps are added to input values by sensors and incremented by explicitly modeled delays. Values are queued at components and consumed in time stamp order. Time stamps also act as deadlines for sending values over the network and for the arrival of values at output components: values are only sent to actuators at the instant given by their time stamps. A program is effectively executed as a discrete-event system linked by synchronized clocks to physical time. While our approach also seeks to implement deterministic reactive behavior in physical time, we focus on a specific mechanism for aligning the time specified in single-node programs with the time that actually passes in implementations. We do not propose to time stamp all values within a program, but rather to allow individual components to specify the 'time stamp' when they must next execute. Valued signals are just an implementation technique to collect deadlines and choose the minimum one to pass to the run-time system.

The concept of Logical Execution Time (LET) [10] also aims to provide an execution semantics that is independent of execution time variations. However, the semantics is delayed, in the sense that inter-thread communication does not occur instantaneously. Conceptually, each thread has its own, monolithic tick function, rather than having one tick function that incorporates the behavior of all threads.

Conceptually, dynamic ticks can implement multiclocking, which also has been explored in Esterel [15], [16]. In multiclocking, it is again typically the environment that governs the execution period, not the application. However, the wake-up times proposed here might be used to implement different clock domains. This might also be raised to higher-level clock specifications, for example in the spirit of the Clock Constraint Specification Language (CCSL) [14].

VI. Conclusions and Outlook

We have presented an approach to augment the determinism of synchronous programming with a flexible approach to incorporate physical time. As illustrated for Esterel, the dynamic tick concept neither requires new languages nor tools; it is rather programming pattern that can be used with many languages and execution platforms. The key idea is to give the application full control over its physical timing context, on the one hand by informing it about the time at the start of a reaction and the duration of the last reaction, and, on the other, by letting it specify when the next reaction should start.

The notion of dynamic ticks that we presented is based on physical time. However, it could also be based on the more general, multiform time mentioned in Sec. I. After all, a clock is just a sensor for a physical parameter, in this case, time; we might also apply dynamic ticks to other run-time parameters, such as the distance traveled by a robot, the number of particles detected, or any other environment variable.

VII. Acknowledgements

The work of von Hanxleden has been supported in part by the German Science Foundation, as part of the Precision-Timed Synchronous Reactive Processing project (PRETSY2, DFG HA 4407/6-2).

References

[1] S. Andalam, P. S. Roop, A. Girault, and C. Traulsen. A predictable framework for safety-critical embedded systems. *IEEE Trans. Comput.*, 63(7):1600–1612, July 2014.

[2] C. André, F. Boulanger, and A. Girault. Software implementation of synchronous programs. In *International Conference on Application of Concurrency to System Design, ACSD'01*, pages 133–142, Newcastle, UK, June 2001. IEEE.

[3] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. In *Proc. IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, Piscataway, NJ, USA, January 2003. IEEE.

[4] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating reactive processes. In *Proceedings of Principles of Programming Languages (POPL'93)*, pages 85–98. ACM, 1993.

[5] Gérard Berry. *The Esterel v5 Language Primer, Version v5_91*. Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 06565 Sophia-Antipolis, 2000. ftp://ftp-sop.inria.fr/esterel/pub/papers/primer.pdf.

[6] V. Bertin, E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venier, D. Weil, and S. Yovine. Taxys = Esterel + Kronos: A tool for verifying real-time properties of embedded systems. In *Proceedings of IEEE Conference on Decision and Control (CDC'01)*, pages 2875–2880, Orlando, Florida, USA, December 2001.

[7] Timothy Bourke and Marc Pouzet. Zélus: A synchronous language with ODEs. In *Proceedings of Hybrid Systems: Computation and Control (HSCC'13)*, pages 113–118, Philadelphia, USA, April 2013. ACM.

[8] Timothy Bourke and Arcot Sowmya. Delays in Esterel. In *SYNCHRON'09—Proceedings of Dagstuhl Seminar 09481*, number 09481 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 22–27 November 2009.

[9] John Eidson, Edward A. Lee, Slobodan Matic, Sanjit Seshia, and Jia Zou. Distributed real-time software for cyber-physical systems. *Proceedings of the IEEE*, 100(1):45–59, January 2012.

[10] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[11] M. Jourdan, F. Maraninchi, and A. Olivero. Verifying quantitative real-time properties of synchronous programs. In *Proceedings of Computer Aided Verification (CAV'93)*, volume 697 of *LNCS*, pages 347–358, Elounda, Greece, June/July 1993. Springer.

[12] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[13] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of Compilers, Architectures, and Synthesis of Embedded Systems (CASES'08)*, Atlanta, GA, USA, October 2008.

[14] Frédéric Mallet and Robert de Simone. Correctness issues on MARTE/CCSL constraints. *Science of Computer Programming*, 106:78–92, 2015.

[15] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, May 2007.

[16] Basant Rajan and R. K. Shyamasundar. Multiclock esterel: A reactive framework for asynchronous design. In *Proceedings of the 14th International Parallel & Distributed Processing Symposium (IPDPS'00), Cancun, Mexico, May 1-5, 2000*, pages 201–210, 2000.

[17] Reinhard von Hanxleden. SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency. In *Proc. Int'l Conference on Embedded Software (EMSOFT'09)*, pages 225–234, Grenoble, France, October 2009. ACM.

[18] E. Yip, A. Girault, P. Roop, and M. Biglari-Abhari. The ForeC synchronous deterministic parallel programming language for multicores. In *International Symposium on Embedded Multicore/Many-core Systems-on-Chip, MCSoC'16*, Lyon, France, October 2016. IEEE.